Improved Stack Allocation Using Escape Analysis in the KESO Multi-JVM

Bachelorarbeit im Fach Informatik

von

Clemens Lang

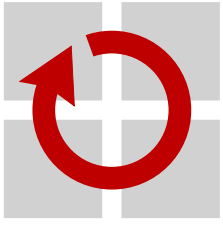
geboren am 09.08.1988 in Lichtenfels

Lehrstuhl für Informatik 4
Friedrich-Alexander Universität Erlangen-Nürnberg

Betreut durch:

Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat
Dipl.-Inf. Christoph Erhardt
Dipl.-Inf. Michael Stilkerich

Beginn der Arbeit: 08. Juni 2012
Ende der Arbeit: 01. Oktober 2012



Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 1. Oktober 2012 _____

Abstract

This thesis describes and evaluates the design and implementation of an escape analysis for KESO, a Java virtual machine for statically configured embedded systems. The goal is to allocate as many objects as possible in stack memory automatically.

Reducing the number of heap-allocated objects in a garbage collected environment can lower the runtime and the complexity and size of the garbage collector's data structures. Especially in hard real-time systems, where algorithms for garbage collection face a set of requirements that are not easily met, this can be worthwhile.

The algorithm implemented in this thesis is a modified version of the one presented by Choi et al. in 2003. In benchmarks, the analysis found 18 to 34 % of allocations to be eligible for stack allocation. Using this optimization, heap memory usage was reduced by up to 29 %.

The results can not only be used for the optimizations implemented yet, but open up a number of further possibilities such as removal of unnecessary synchronization primitives. Potential for optimization can also be found in methods allocating an object and returning a reference to it. Instead of using heap memory for the returned object, the caller be modified to pass a reference to a sufficiently large chunk of memory in its stack frame, which the callee could use for the object to be returned.

Zusammenfassung

Diese Arbeit beschreibt und evaluiert Entwurf und Implementierung einer Fluchtanalyse für KESO, eine virtuelle Maschine für Java für statisch konfigurierte eingebettete Systeme. Das Ziel ist die automatische Allokation möglichst vieler Objekte im Stapelspeicher.

Die Anzahl der in der Halde allokierten Objekte zu reduzieren kann in einer Umgebung mit automatischer Speicherbereinigung sowohl deren Laufzeit als auch die Komplexität und Größe ihrer Datenstrukturen senken. Vor allem in harten Echtzeitsystemen, in denen Algorithmen zur automatischen Speicherbereinigung eine Reihe von schwierigen Anforderungen erfüllen müssen, kann dies lohnenswert sein.

Der Algorithmus, der in dieser Arbeit implementiert wurde, ist eine Adaption einer Veröffentlichung von von Choi et al. 2003. In Tests konnten zwischen 18 und 34 % aller Allokationen als stapelallokierbar identifiziert werden. Durch diese Optimierung sank die Auslastung des Haldenspeichers um bis zu 29 %.

Die Ergebnisse können nicht nur für die bereits implementierten Optimierungen verwendet werden, sondern eröffnen eine Reihe weiterer Möglichkeiten, wie z. B. das Entfernen unnötiger Synchronisationsprimitiven. Optimierungspotential besteht auch in Methoden, die ein Objekt anlegen und eine Referenz darauf zurückgeben. Anstatt für das zurückgegebene Objekt Speicher aus der Halde zu verwenden könnte der Aufrufer modifiziert werden eine Referenz auf ein ausreichend großes Stück Speicher aus seinem Stapelbereich weiterzugeben, das vom Aufgerufenen für das zurückzugebende Objekt genutzt werden könnte.

Contents

1 | Introduction

Our daily lives have become pervaded with embedded systems: the digital alarm clock that wakes you in the morning, the coffee machine using microcontrollers to accept your push of a button and control the water flow, your microwave oven, maybe even your light switches – they all contain programmable electronics tailored to the specific needs of the device they are built into.

Traditionally, the microchips in these devices are dedicated to a single purpose and programmed in C and kindred languages, or even assembly language. Technical progress has increased the computational power of these chips and initiated an ongoing trend to integrate multiple features into a single chip to achieve higher efficiency. This has created a desire to consolidate several tasks on the same chip and use the available processing power for increasingly complex tasks. These changes pose a new set of problems previously unknown in embedded systems development.

Isolation Multiple tasks on the same chip should be isolated to prevent them from interfering with each other. This is common in multitasking desktop operating systems and usually accomplished by using virtual address spaces. Because embedded systems are built for high efficiency – in terms of chip area, energy consumption and production costs – they often lack hardware-based memory protection, a requirement for virtual address spaces, completely or only support it rudimentarily. KESO addresses this problem using a combination of type safety in the source language (Java) and runtime checks to prevent accessing unrelated memory possibly used by other tasks.

Ease of use Developing large software systems in low-level programming languages is impracticable. Languages allowing higher productivity are at advantage when writing large-scale systems. The Java programming language is an attractive choice, because it offers safety, low maintenance costs and wide availability of developer tools and well-trained developers. “Java was developed to make code development cleaner and more bug free [...] Java virtual machines, even the just-

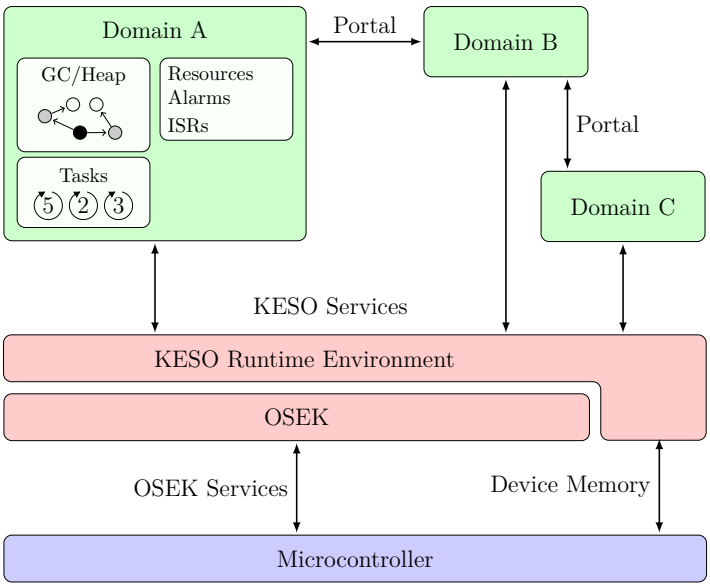


Figure 1.1: Schematic overview of the KESO system at runtime

in-time and ahead-of-time compiled versions, are still too big and too slow for use on most microcontrollers" [Col12], others say about Java in embedded systems. KESO compensates for the performance penalty using an aggressively optimizing compiler and assumptions about the nature of embedded systems.

1.1 | The KESO Multi-JVM

KESO is an application-tailored Java virtual machine for statically configured embedded systems. It provides an abstraction layer on top of an OSEK/VDX or AUTOSAR real-time operating system. Applications and even device drivers for KESO are written in Java. The KESO compiler *JINO* tries to exploit static knowledge about software and system configuration to perform aggressive optimizations in order to overcome the performance penalty of Java. KESO-based systems cannot load further code or create new tasks at runtime. This allows *JINO* to use much more aggressive optimizations, because the whole codebase is known and can be optimized at compile-time.

In comparison to traditional virtual machines, KESO does not use bytecode interpretation or just-in-time compilation, but transforms Java bytecode to standard C (or C++) ahead of time. This approach makes execution speeds comparable to software written in C possible.

Figure 1.1 depicts the schematic architecture of a KESO-driven system. An abstraction layer on top of the operating system's API is provided by the KESO runtime environment. It also allows executing native code using a mechanism similar to the Java native interface (JNI) as well as access to configurable locations in the address space – e.g., for memory-mapped I/O. Within this environment, the system supports multiple so-called "domains". Domains provide a similar isolation level as address spaces do on UNIX systems. Each domain can be seen as separate Java virtual machine, which makes KESO a "Multi-JVM". A domain has its own heap and garbage collector, defines own resources, alarms and interrupt service routines and may have several tasks. Passing data between domains is only possible through well-defined gates, the so-called "portals". The portal mechanism ensures that the isolation property cannot be violated.

1.2 | Motivation

While Java's automatic memory management makes writing software easier, its need for garbage collection may have a considerable impact on execution speed. This becomes obvious when comparing the different strategies for memory allocation.

Explicit memory allocation and de-allocation using library functions have been the standard dynamic memory management method in C and C++. While it is relatively simple from a user's point of view, management of the fragmented memory chunks introduces medium complexity into these library functions. De-allocation is simple and fast and happens as soon as the programmer releases the memory. The downsides are manifold: Programming errors can lead to memory leaks (i.e., memory that cannot be reclaimed although it is no longer in use) or even security vulnerabilities, e.g. by using memory after releasing it or attempting to deallocate memory multiple times.

A different approach is using automatic garbage collection. It removes the need for (and often also the possibility of) manual freeing of memory. Instead, automatic scans identify unreachable objects and reclaim the associated memory. This task is complex and time-consuming, since it needs to scan all objects in use. Using garbage collection does not free memory immediately after its last use, but defers de-allocation to the next run of the garbage collector. Compared to manual memory management, garbage collection is less error-prone at the cost of being slower at reclaiming memory and less predictable.

Apart from manual memory management and the use of a garbage collector, region-based memory management is a third alternative. Allocating (and de-allocating) objects from regions is simple and quick, because regions are only reclaimed as a whole. This limitation speeds up region-based allocation operations at the cost of dead objects possibly not being reclaimed while the region is still

active. The Real-Time Specification for Java formulates a variant of region-based memory called scoped memory [Dib06].

At last, memory can also be acquired on the stack by moving the stack pointer. This method of memory management is the cheapest, since allocation and reclaiming are only arithmetic operations on a CPU register. Memory allocated on the stack is however not as flexible as memory from heap: Because returning from a method will also automatically reclaim the memory allocated on its call stack frame, objects that exceed the lifetime of the method in which they are created, cannot be allocated on the stack.

Since stack memory is allocated and reclaimed a lot faster than Java's traditional garbage collection could ever achieve, turning allocations into stack-allocations where possible can benefit the performance of applications. Especially those with intensive dynamic memory allocation behavior should receive a performance boost by using stack-allocations where possible.

Rather than having a programmer decide which objects can be allocated on stack manually, a compiler could identify local objects automatically. To compute the required information, the compiler first needs to identify which references can point to which objects at any given point in the program. Optimizing compilers usually already have this information readily available, because it was gathered in alias analysis. Second, determining reachability from references that are known to be escaping the method where they were obtained decides whether an object in the graph is local or not. These steps are also called escape analysis. Since escape analysis needs information also computed by alias analysis, many techniques that are known from algorithms for alias analysis also apply.

This thesis strives to implement an analysis and transformation pass and changes to the backend for KESO's *JINO* compiler that will allocate objects not escaping their method of creation on the stack without intervention of the programmer. This should relieve of the strain on the garbage collector and reduce the frequency of garbage collection. It could also reduce maximum heap usage in phases of the program with intensive dynamic memory allocation and accelerate execution of the software depending on the complexity of allocation operations.

1.3 | Document Structure

The following chapter describes the state of the *JINO* compiler before this thesis and similar previous work implemented in *JINO*. Chapter 3 explains the algorithms used for escape analysis and stack allocation. In Chapter 4, the implemented optimization is benchmarked and graded, before the last chapter concludes and lists future work based on findings in this thesis.

2 | State of the Art

The KESO Multi-JVM does not interpret or JIT-compile Java bytecode on the target processor. Instead, it compiles application code to native machine code ahead of time. In this step, it optimizes the code and minimizes the runtime environment's size using static knowledge about the configuration. This makes virtual machine environments created by KESO highly customizable and tailored to the needs of the application. The current chapter gives an overview of the build tool used to achieve this with a focus on the parts relevant for this thesis.

2.1 | The JINO Compiler for the KESO Multi-JVM

The *JINO* compiler is an integral part of the KESO system. Similar to most modern compilers, it is modularized into a language-specific frontend, a series of optimizing passes working on intermediate code and a number of target-specific backends generating code from the intermediate representation.

Frontend *JINO*'s frontend only supports Java code. It uses the Java Compiler API [Ora11] to generate a bytecode representation of the source and parses the resulting `.class` files to build *JINO*'s intermediate code representation. The frontend also reads the configuration needed for application-tailored optimization.

Middle-End The intermediate tier (or middle-end) contains all optimizations implemented in the *JINO* compiler. It exclusively uses the intermediate code representation constructed in the frontend. *JINO*'s intermediate code bears resemblance to Java bytecode in its instruction set, but is fundamentally different from said bytecode in the way that is not stack-based but explicitly references its operands. Rather than specifying names of virtual registers, this representation points to the results of previous instructions using a reference to the computing instruction, which creates a tree-like structure. This simplifies separating the analysis of right-hand

```
public static int factorial(int);
  Code:
    0:   iload_0
    1:   iconst_1
    2:   if_icmpgt    7
    5:   iconst_1
    6:   ireturn
    7:   iload_0
    8:   iload_0
    9:   iconst_1
    10:  isub
    11:  invokestatic     #2; //Method factorial:(I)I
    14:  imul
    15:  ireturn
```

Listing 2.1: Java bytecode of a method computing the factorial

and left-hand sides in assigning statements as used in Section 3.1.2. See Listing 2.1 and Listing 2.2 for an example of how Java bytecode maps to *JINO*'s intermediate code (note that the textual representation of the intermediate code uses numerical virtual registers; those are only used when dumping the intermediate code, though).

The middle-end contains a number of transformations commonly found in optimizing compilers, such as

- SSA construction and deconstruction [Erh11, 3.5],
- liveness analysis and removal of dead variables and their assignments [Erh11, 3.4],
- removal of unreachable methods [Erh11, 3.7.1] and dead code [Erh11, 3.7.2],
- elimination of redundant runtime checks [Erh11, 3.8],
- constant and copy propagation [Erh11, 2.2.2] and
- method inlining [Erh11, 2.2.2].
- transformation of unambiguous virtual calls into their non-virtual counterparts [Erh11, 3.7.1],
- removal of unused fields [Erh11, 3.7.3, 2.2.2] and
- optimization of runtime lookups if the result is known at compile-time [Erh11, 3.9].

It is worth noting that some of these optimizations are only worthwhile because *JINO* may assume a closed world scenario, i.e., it can analyze the complete codebase at compile-time. For example, public or protected fields or methods could never be considered unused in traditional Java according to the Java Language Specification [GJS+12], because dynamically loaded code (possibly unavailable to analyze at compile-time) might access the field or call the method in question. For similar reasons, virtual method calls can not always be transformed into non-virtual

```
factorial(I)I {
_B0: (done; LiveIn: [i0]; PhiIn: []; LiveOut: [i0])
      0:     %0 = IReadLocalVariable i0
      1:     %1 = IConstant 1
      2:     %2 = GTConditionalBranch %0, %1, [_B5, _B7]
_B5: (done; LiveIn: []; PhiIn: []; LiveOut: [])
      6:     %3 = Goto [_B17]
_B7: (done; LiveIn: [i0]; PhiIn: []; LiveOut: [i1_1])
      8:     %4 = IReadLocalVariable i0
      9:     %5 = IConstant 1
     10:     %6 = ISub %4, %5
     11:     %7 = IStoreLocalVariable i2_0, %6
     11:     %8 = IReadLocalVariable i2_0
     11:     %9 = InvokeStatic test/Factorial.factorial(I)I %8
     14:    %10 = IStoreLocalVariable i2_1, %9
      7:    %11 = IReadLocalVariable i0
     14:    %12 = IReadLocalVariable i2_1
     14:    %13 = IMul %11, %12
     15:    %14 = IStoreLocalVariable i1_1, %13
     15:    %15 = Goto [_B17]
_B17: (done; LiveIn: []; PhiIn: [i1_1]; LiveOut: [])
     -1:    %16 = IReadLocalVariable i1_1
      5:    %17 = IConstant 1
     -1:    %18 = Phi %16 [_B7], %17 [_B5]
     -1:    %19 = IStoreLocalVariable i1_2, %18
     17:    %20 = Epilog i1_2
}
```

Listing 2.2: *JINO* intermediate code generated from Listing 2.1.

calls, because new candidates might be added at runtime. *JINO* will also propagate constant parameters into methods if all call sites use the same constant. Since loading new code possibly adds new invocation sites, this optimization would be illegal without the closed-world assumption.

See [ESLSP11] for a case-study of optimizations that are only possible in statically configured systems.

It would be best to implement the analysis of the escape property in this phase of the compiler.

Backend Multiple target-specific code emitters make up the backend. Since code generation for allocations is done in this phase, transforming allocations into stack allocations should likely be implemented on this level. Aside from generating C code, the backend also writes an *OSEK Implementation Language (OIL)* file for the OSEK/VDX system generator to allow building an application-tailored kernel.

2.2 | JINO's Pass Model

To organize analysis and transformation steps and their relations, *JINO* employs a pass model inspired by the Low-Level Virtual Machine (LLVM) compiler infrastructure[1]. Analyses and transformations are implemented as passes whose execution order is determined by the pass manager based on dependencies and anti-dependencies and a flag that allows disabling the pass. Since fixpoint iteration is commonly used in compilers, a flag to loop while the pass signals that it needs to be run again is offered as a convenience. A pass also holds information which results of earlier passes it invalidates and thus would require re-calculation if they were to be used again. Similar to LLVM's passes, *JINO* offers a number of abstract base classes for passes iterating over domains, classes or methods for developers to extend. [Erh11, LLV]

Construction of static single assignment (SSA) form and computation of the dominator tree can serve as example for the pass model: SSA construction uses information about the dominance frontier and thus has a dependency on dominator tree computation. Transforming the program into SSA form does not change the dominator tree; for this reason, SSA construction will not mark the dominator tree outdated, allowing re-use rather than re-calculation. Other passes require the code to be SSA-formed and thus declare a dependency on SSA construction and an anti-dependency on SSA deconstruction, ensuring these passes will be run between construction and deconstruction of SSA form. A number of optimizations can be selected by the user at compile-time. The pass manager computes the execution order based on the passes requested by the user and their dependency relations.

[1] `http://llvm.org/`

Escape analysis can be implemented as a pass in the *JINO* compiler. Since we want to compute the escape property on intermediate code in SSA form, the pass needs to depend on its construction and anti-depend on its deconstruction.

Because stack-allocation of function-local objects happens in the backend, it cannot be implemented easily as a pass. However, a pass can instead annotate candidate allocation operations in intermediate code for the backend to interpret. Since this pass needs the information collected in escape analysis, it needs a dependency on the analysis pass.

2.3 | Existing Escape Analysis

JINO already features a previous implementation of this kind of analysis, albeit based on a different algorithmic idea. It runs over the intermediate code representation trying to trace paths of memory locations across assignments, method invocations and other relevant statements. If a variable is found to be passed to a reference known to escape a method (e.g., because it is a parameter) or to a statement causing the escape of the value (e.g., `throw` and `return`), it is marked escaping. The analysis makes a number of simplifying assumptions and stops tracking memory location at a couple of statements like assignments to members or array fields.

The proposed implementation of escape analysis will not make any of these simplifying assumptions. It will also not only generate boolean information as to whether an object escapes or not, but a ternary state divided into local objects (those that do not escape), method-escaping objects (escaping their method, but not thread of creation) and global objects (escaping both the method and thread of their allocation). This partition enables further optimizations like synchronization optimization or conversion into a “caller allocates” pattern for objects marked *method*. See Chapter 5, where this is covered in more detail.

3 | Design and Implementation

Objects, whose lifetime does not exceed that of the scope in which they were created, can be allocated on the stack. Doing so simplifies both their allocation and deallocation procedure, because stack-based memory is allocated faster and often reclaimed earlier than memory from the heap managed by a garbage collector. To determine this property, alias information[1] is needed. While alias information can be computed locally, results yielded by global analysis are much more accurate. Since the architecture of KESO does not allow unknown code to be called, no assumptions have to be made about the implementation of such unknown functions[2].

This chapter explains the algorithm and implementation of said alias analysis and stack allocation transformation. The implemented algorithm is based on [CGS+03].

3.1 | Intraprocedural Analysis

A connection graph (CG) as defined in [CGS+03, section 2] is used to store the required alias information. The algorithm to build the CG consists of two separate phases. First, function-local results are computed in an intraprocedural analysis. The part of the calculated graph relevant for other methods, the so-called non-local subgraph is then used in an interprocedural pass to derive complete alias information. Reachability in the CG decides whether an allocation can be transformed into a stack allocation without violating the correctness of the program. [CGS+03]

[1] alias information is the information which object references can – at any point in time in the runtime of the program – point to a certain object

[2] with the exception of native code, see Section 3.3.2

3.1.1 | The Connection Graph

The CG used to represent the alias information is a graph consisting of several different vertex and edge types. The following sections describe these vertices and edges in detail.

3.1.1.1 | Nodes in the Connection Graph

Vertices in a CG are called "nodes" and are either object nodes or reference nodes.

Object nodes represent an object, i.e., an instance of a class. It is important to note that the analysis creates at most one object node per allocating statement. Keep in mind that an allocation might be executed multiple times at run-time, but is still represented by the same object node in the CG [CGS+03, p. 879]. Some of the objects in the program do not have an allocation site, e.g., string constants. Those are tracked in a subclass of object nodes called constant object node. At some points in the program, the algorithm requires an object node, but the information available is not yet sufficient to determine its allocation site. This happens when trying do dereference a reference, but it is not known at this point in the analysis where the reference points to. It is usually the case if the reference's pointees were created outside of the analyzed method and the reference was passed in via a parameter, or if the reference is always `null`. The algorithm conservatively assumes the former, which might cause the addition of a superfluous edge, but otherwise does not affect the correctness of the analysis [CGS+03, p. 883]. To represent the pointees of the dereferenced reference in these cases, a phantom object node is created. At most one phantom object node is created per intermediate code instruction.

Reference nodes are created for every reference used in the analyzed program, e.g., references in local or global variables, or references returned from function calls. There are four subgroups of reference nodes:

CG Local Reference Node Local reference nodes represent references stored in local variables, i.e., slots in Java bytecode.

CG Global Reference Node Global variables[3] of reference type cause the creation of a global reference node in the CG.

CG Field Reference Node Member variables of objects are depicted in the CG by adding a field reference node and a field edge pointing from the object node to the newly added reference node. Only fields of reference type are of interest; others are simply ignored. Field reference nodes are annotated with the field's name in the class of the object node.

CG Actual Reference Node These nodes are added for every formal parameter or return value of reference type both on the caller and on the callee side.

[3] i.e., static class members in Java

node type	object node	field reference node	reference node
object node	—	field edge	—
reference node	points-to edge	deferred edge	deferred edge

Table 3.1: Type of edge depending on the type of the source node (down) and destination node (across). "—" denotes impossible combinations. This is implemented as the `attach` operation.

3.1.1.2 | Edges in the Connection Graph

Nodes are interconnected with a series of different edges. Edges never have any duplicates, that is, a CG never contains more than one edge with the same combination of origin and destination node.

A points-to edge from a reference node to an object node represents the information that the reference node can point to a object node at some point in the program.

To simplify updating the CG with new information, deferred edges are temporarily used: At every point where a reference is copied, rather than copying all existing points-to edges in the CG, a deferred edge is added from the copy operation's destination's representation in the CG to its equivalent of the copy operation's source. After intraprocedural analysis these nodes are removed in a path compression step (see Section 3.1.5).

Field edges point from object nodes to field reference nodes when the class of the object node has a member variable of reference type identified by the name in the destination field reference node.

To simplify adding edges in the CG a helper function `attach` was implemented to aid in creating the correct edge type depending on the types of the source and destination node. Its behavior is shown outlined in Table 3.1.

3.1.1.3 | Escape State

Analogously with [CGS+03, p. 881], the escape property of objects from methods and threads is defined as follows: Let O be an object and M be a method invocation. O escapes M if its lifetime exceeds the lifetime of M. Let T be a thread. O escapes T if it is reachable from any other thread $T' \neq T$.

To determine the escape state of an object, the escape property is extended to all nodes in the CG, i.e., each node is annotated with its escape property. Possible values for the escape state are (1) *local*, denoting the node does neither escape its method of creation nor the thread running this method, (2) *method*, used if the

node escapes its method of creation, but not the thread (or task in a KESO context) running this method, and (3) *global* indicating the node escapes both the method and thread of its creation. These states can be grouped in a total order given by *local* < *method* < *global*, because a node with a lifetime exceeding the lifetime of the thread it was created in will also exceed the lifetime of the method it was created in (i.e., *global* includes *method*).

The algorithms start by assuming no node escapes the method or thread of its creation, i.e., by setting the escape state to *local*. Performing reachability analysis in the CG allows determining the escape state of nodes as follows: any node N' reachable from a given node N must fulfill

$$\text{escapeState}(N') \geq \text{escapeState}(N) \tag{3.1}$$

For any node found which does not satisfy Equation (3.1), update its escape state as follows:

$$\text{escapeState}(N') = \text{escapeState}(N) \tag{3.2}$$

Nodes reachable from a node with *global* escape state are also marked globally escaping; nodes reachable from a method-escaping node, but not reachable from a global context get an escape state of *method*. All other nodes remain at the default after creation, which is *local*.

3.1.2 | Building the Connection Graph

The CG is constructed by iterating over the intermediate code instructions of every method in SSA form. For each method, a new CG is created and nodes and edges are added as follows.

At the beginning of a method, the algorithm creates a callee-side actual reference node and a local reference node for each formal parameter of reference type and adds a deferred edge from the local reference node to the actual reference node. If the analyzed method returns a reference, it also adds an actual reference node representing the return value. For each actual reference node created on the callee side, its escape state is set to *method*.

To further build the CG, the algorithm iterates over all instructions in a method, ignoring all but the following operations.

3.1.2.1 | Local Variables

Assignments to local variables of the form $p = q$ are handled by creating a local reference node to represent p in the CG if none exists for this variable yet and adding an edge (by means of `attach` as described in Table 3.1) from p to q[4]. This

[4]or rather, their representations in the CG

makes no assumptions about the nature of q: it might be another local variable, a field, or even a method invocation. See the appropriate parts of Section 3.1.2 on how to handle the right-hand side in an assignment.

Reading a local variable causes the creation of a local reference node for the local variable if none exists yet. Further handling of the local reference node depends on the operation that uses the read value.

3.1.2.2 | Global Variables

Usage of a global variable as lvalue[5], e.g., $T.p = q$, causes the creation of a global reference node if none exists for $T.p$ yet and the addition of an edge from this node to q (where the edge type depends on the type of q). For every global reference node created, its escape state is set to *global*.

Global variables as rvalues[6] also cause the 1-limited creation of a corresponding global reference node. Given a statement like $q = T.p$ where $T.p$ is the global variable, an edge from q to the global reference node representing $T.p$ is added (where again, the type of the newly added edge is determined by the `attach` operation).

3.1.2.3 | Allocations

Object allocations like $p = new\ T()$ cause the creation of an object node to represent the newly created instance of T. At most one object node per allocating instruction shall be created. Allocations of arrays[7] are handled in the same manner.

Constant references (e.g., constant strings and `null` constants) cause the creation of a constant object node and are otherwise processed analogously.

The remaining assignment in $p = new\ T()$ is handled depending on the type of p. E.g., if p is a local variable, see Section 3.1.2.1.

3.1.2.4 | Fields and Arrays

Instructions involving field access[8] in the form of $q = p.f$ or $p.f = q$ (where p is a reference to an object holding the field and f is the name of the field being accessed) are handled as follows:

(1) For p and each reference node reachable from p via deferred edges, make sure the pointees of this reference node are represented in the CG. This is the case if the node has any outgoing edges. Thus, for reference nodes without any

[5] via the Java bytecode instruction `putstatic`
[6] via the Java bytecode instruction `getstatic`
[7] Java bytecode instructions `anewarray`, `multianewarray`, `newarray`
[8] Java bytecodes `getfield` and `putfield`

outgoing edge, create a phantom object node and attach it to this reference node. This might create unnecessary phantom object nodes to temporarily hold alias information, which will be removed later in Section 3.1.5.

This ensures the pointees of p are represented in the CG, because they will be needed in step 2.

(2) For each object node reachable from p by deferred edges and points-to edges, make sure a field reference node attached to the object node exists for the to-be-accessed field. If necessary, create this field reference node.

Compile a list of the field reference nodes found for step 3.

(3) For each field reference node found in step 2, do the following:

(3a) If the field is written to, i.e., $p.f = q$, attach q to the field reference node.

(3b) If the field is read from, i.e., $q = p.f$, attach the field reference node to q.

Reading and writing to arrays is handled analogously with field access, i.e., $q = p[i]$ is handled as $q = p.\alpha$ and $p[i] = q$ is handled as $p.\alpha = q$, where α is a special identifier that can not occur in field operations. The algorithm currently does not distinguish between different array indices (although that could be done by using a mapping function $\alpha(index)$ and assuming writing or reading from all indices if the index is not known at compile time).

3.1.2.5 | Φ-Functions

Since the analysis is run on code in SSA form, the code may contain Φ-functions. To correctly handle them, ensure a local reference node exists for the result of the Φ-operation and loop through the candidates for the variable, attaching each candidate to this local reference node.

If the code was not SSA-formed, special care would have to be taken when processing assignments and when reaching join points of the control flow. Variable re-assignments would have to cause any incoming deferred edges to be redirected to the successors of the reference node before processing the new assignment. To correctly process alternative control flows assigning the same variable, the CG would either have to be copied or the changes made to local variables in one of the paths would have to be reverted before processing the other. Both approaches require merging the information at control flow join points [CGS+03, p. 885]. Using SSA form, all CG updates can be done in place.

3.1.2.6 | Exceptions

Throwing exceptions causes the thrown object to escape the method of its allocation. In order to represent this information in the CG, *throw p* statements[9] need to set *p*'s escape state to *global*.

3.1.2.7 | Method Calls

For any method invocation in the form of $p = q.method(r, s, t)$ encountered, create a caller-side actual reference node for each formal parameter of reference type and `attach` it to the representation of the actual parameter given at this invocation site. The `this`-parameter is implicitly treated as first parameter and handled like other parameters.

If the called method returns a reference, also create a caller-side actual reference node to represent the return value. The further processing of this node depends on whether the return value is used and how it is used. In the example given above, an edge would be added from the representation of p to this actual reference node.

3.1.2.8 | Return Statements

Returning statements not returning a reference are ignored. Those returning a reference of the form *return p* are handled by adding a deferred edge from the actual reference node representing the return value (which was created in Section 3.1.2) to *p*.

3.1.3 | Reachability Analysis

To determine which objects in the analyzed method can be stack-allocated, reachability analysis is performed on the CG of the method. All nodes reachable from a node with an escape state of *global* have their escape state set to *global*, too. Nodes reachable from a node with an escape state of *method* are marked with the same escape state if their escape state is not *global*.

This step can be done gradually during the creation of the CG by modifying `attach` to also update the escape state of any new dependents.

3.1.4 | Example

Consider the simple Java class given in Listing 3.1. When compiled to Java bytecode, the code of the `test` method looks like Listing 3.2. Note that this example already

[9] Java bytecode `athrow`

```
1  public class Test {
2    public static Object a;
3    public Object b;
4
5    public static String test(String arg) {
6      Test t = new Test();
7      t.b = arg;
8
9      Object o = new Object();
10     Object p = o;
11
12     Test.a = p;
13
14     String s = "Hello, World!";
15     return s;
16   }
17 }
```

Listing 3.1: Example Java code for interprocedural analysis

```
public static java.lang.String test(java.lang.String);
    0:  new #2; //class Test
    3:  dup
    4:  invokespecial #3; //Method "<init>":()V
    7:  astore_1
    8:  aload_1
    9:  aload_0
   10:  putfield  #4; //Field b:Ljava/lang/Object;
   13:  new #5; //class java/lang/Object
   16:  dup
   17:  invokespecial #1; //Method java/lang/Object."<init>":()V
   20:  astore_2
   21:  aload_2
   22:  astore_3
   23:  aload_3
   24:  putstatic #6; //Field a:Ljava/lang/Object;
   27:  ldc #7; //String Hello, World!
   29:  astore  4
   31:  aload 4
   33:  areturn

}
```

Listing 3.2: Listing 3.1 compiled to Java bytecode

is in SSA form, because each variable is only assigned once.

To create the CG for this method, start by creating a callee-side actual reference node for the parameter `arg` in slot 0 and setting its escape state to *method*. In Figure 3.1 the created node is denoted by $arn : arg$. Note that no node for a `this`-parameter is created, because the method is static and thus has no such parameter. Further, also create a local reference node for the variable in slot 0, denoted by arg in the graph and add a deferred edge from this node to its corresponding actual reference node to simulate an assignment.

Since the method returns a reference (a string in this case), create an actual reference node for the return value *ret* and set its escape state to *method*.

For the following object allocation, an object node is created.

The next relevant instruction is the invocation of `Test`'s constructor. Create an actual reference node for each formal parameter of reference type, i.e., `this`, and add a points-to edge from the created actual reference node, which is denoted by *Object.init(this)* in the example, to the actual argument, i.e., the allocated object node. Since the constructor does not return a reference, do not create a caller-side actual reference node for the return value.

In instruction 7, the reference to the allocated `Test` object is stored into a local variable in slot 1. This causes the creation of a local reference node for this slot (t in Figure 3.1) and the addition of a points-to edge from t to the object node.

The two following `aload` instructions do not modify the CG, because local reference nodes for these variables already exist in the graph. For the `putfield` instruction at position 10, create a field reference node and attach it to the object node pointed to by t, since none exists for b yet. Finally, process the assignment of arg to b by adding a deferred edge from b to arg.

Instructions 13 – 20 are handled similarly to the object allocation in 0 – 7.

Bytecode positions 21 and 22 code the copy operation $p = o$. Since no local reference node exists for p yet, a new one is created. A deferred edge is added from p to o to represent the assignment.

`Putstatic` marks a write to a global variable. No global reference node exists for the field $Test.a$ yet, so a new one is created and its escape state is set to *global*. The assignment from slot 3 to the global variable is handled by adding a deferred edge from $Test.a$ to the variable in slot 3, p. This causes all nodes reachable from p to be marked with a *global* escape state (either in reachability analysis after building the graph, or immediately if `attach` propagates escape information).

Instructions 27 – 29 load a string constant into a slot. According to Section 3.1.2.3, constants are handled like allocations, and thus create a constant object node for the string. The following assignment to the local variable in slot 4 causes the creation of a local reference node, denoted by s in Figure 3.1, and a points-to edge from s to the constant object node.

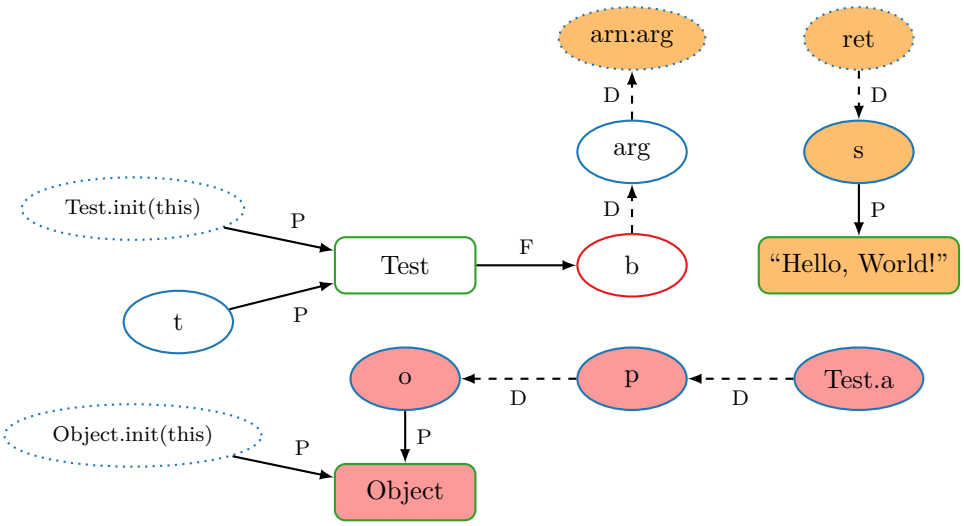


Figure 3.1: The CG of the method given in Listing 3.1 after interprocedural analysis but before path compression. Elliptic vertices represent reference nodes. Field reference nodes have a red • border, other reference nodes have blue • borders. Dotted borders mark actual reference nodes. Vertices with a rectangle shape and green • border are object nodes. Rectangles with dashed borders (not present in this graph) are phantom object nodes.
Deferred edges are dashed and annotated with the letter "D". "P" marks points-to edges and "F" is used for field edges.
The escape state of nodes in the CG is denoted by the fill color. White indicates *local*, orange • stands for *method* and red • marks *global*.

Finally, statements 31 – 33 require a deferred edge to be added from *ret* to *s* in the CG. This also causes the propagation of *ret*'s escape state *method* to its descendants *s* and the constant object node.

3.1.5 | Graph Compression

At the end of intraprocedural analysis the resulting graph can be compressed by removing all deferred edges. An incoming points-to edge can always be removed from the CG by replacing it with edges to successor nodes. The type of the new edge is equal to the type of the edge pointing to the successor node. If a reference node does not have any outgoing edges (i.e., it is a "terminal" node), create a phantom object node and attach it to the node (which adds a points-to edge). Applying this algorithm recursively will remove all remaining deferred edges from

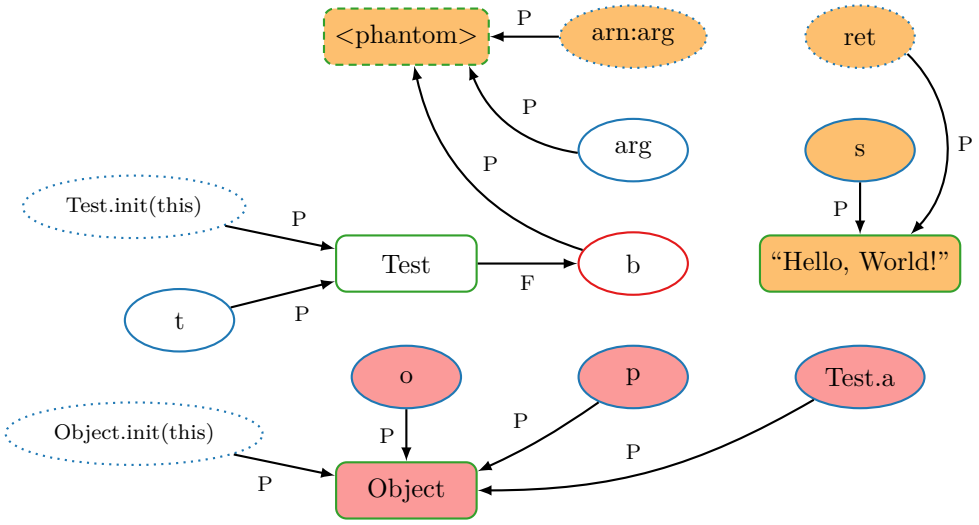


Figure 3.2: The CG given in Figure 3.1 after path compression. Colors and shapes as described in Figure 3.1 with the addition of phantom object nodes with dashed green • border.

the CG [CGS+03, p. 880]. Figure 3.2 depicts the example given in Section 3.1.4 after applying path compression.

The graph can be simplified further by removing all phantom object nodes which are siblings of any other object node to remove any phantom object nodes unnecessarily added in Section 3.1.2.4. If the phantom object node to be removed has any outgoing field edges, re-create the field reference nodes reachable via these field edges and their dependents below all sibling object nodes of the phantom object node. Figure 3.3 illustrates this step.

3.1.6 | Interim Results

The results generated in the intraprocedural analysis are only final for nodes that are not reachable from callees of this method. Consider Listing 3.3 for a case where the local alias information is not sufficient to determine whether a node can be allocated on the stack: The newly created `ListElement` in `addElement` would be marked *local* by intraprocedural analysis, but is passed as argument to `appendAfter` (i.e., it is reachable from a callee). If it was stack-allocated, it would no longer be available after the lifetime of the `addElement` method, which contradicts the principle of a linked list.

Global analysis is needed to convey the information that an edge is added from

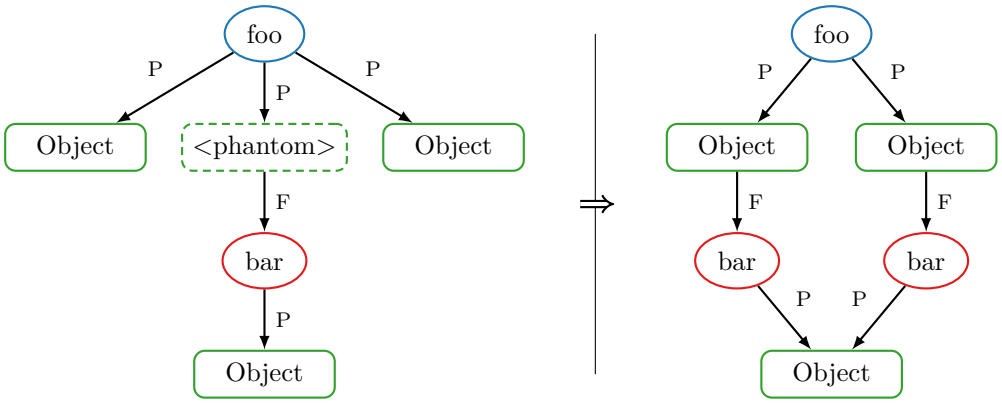


Figure 3.3: Removal of superfluous phantom object nodes (see Section 3.1.5). Colors and shapes as described in Figure 3.2.

```
1  public class LinkedList {
2    private static final class ListElement {
3      ListElement next;
4      Object elem;
5
6      public ListElement(Object o) {
7        elem = o;
8      }
9    }
10
11   private ListElement head;
12
13   public static void addElement(LinkedList l, Object o) {
14     appendAfter(l.head, new ListElement(o));
15   }
16
17   private static void appendAfter(ListElement pred, ListElement succ) {
18     pred.next = succ;
19   }
20 }
```

Listing 3.3: Example where intraprocedural analysis does not generate sufficient information

`l.head.next` to the new list element by the invocation of `appendAfter`. Section 3.2 describes further analysis solving this problem.

3.2 | Interprocedural Analysis

To solve the problem outlined in Section 3.1.6, a global analysis, i.e., an analysis spanning multiple methods, is needed. In this step, the summary information calculated for a method in intraprocedural analysis is used to update the CG of any callers. The algorithm described in [CGS+03, section 4] allows using a single CG to represent the effects of a method independently of the various possible calling contexts. The relevant part of the CGs calculated in intraprocedural analysis are the nodes with an escape state of *global* or *method*, the so-called non-local subgraph.

Information needs to be propagated from a callee's CG to the CG of its callers, i.e., bottom-up in the call graph. Since this step might modify the caller's CG its callers would have to be updated in turn. To prevent recalculation of information, this should be done bottom-up in a topological order. Because recursion will cause cycles in the call graph, such a topological order does not always exist. Fixpoint iteration could solve this problem, but is very expensive when applied to the whole graph. Instead, iterating over each strongly connected component in the graph separately until its partial solution converges and processing other vertices in bottom-up topological sorting gives acceptable runtime. To find the strongly connected components and sort them topologically, Tarjan's algorithm [Tar72] is used, because it computes both the strongly connected components and the sorting simultaneously.

3.2.1 | Node Propagation

Each invocation in the currently analyzed method is processed as follows: Starting with each pair of caller-side and callee-side actual reference nodes $(\hat{a}, a)$ recursively determine further equivalence pairs between caller and callee and ensure nodes and edges between them that are present in the callee CG are also represented in the caller CG.

Consider the example given in Listing 3.3 where intraprocedural analysis did not generate complete information. The CGs for `addElement` and `appendAfter` are given in Figures 3.4 and 3.5, respectively. There are two method invocations in `addElement`: (1) the constructor of `ListElement`, which will not be discussed in this example for simplicity reasons, and (2) the call to `appendAfter`. The procedure to update the caller CG with the information from `appendAfter` would start at the tuple

$$\big(\text{appendAfter(pred)},\ \text{arn:pred}\big) \tag{3.3}$$

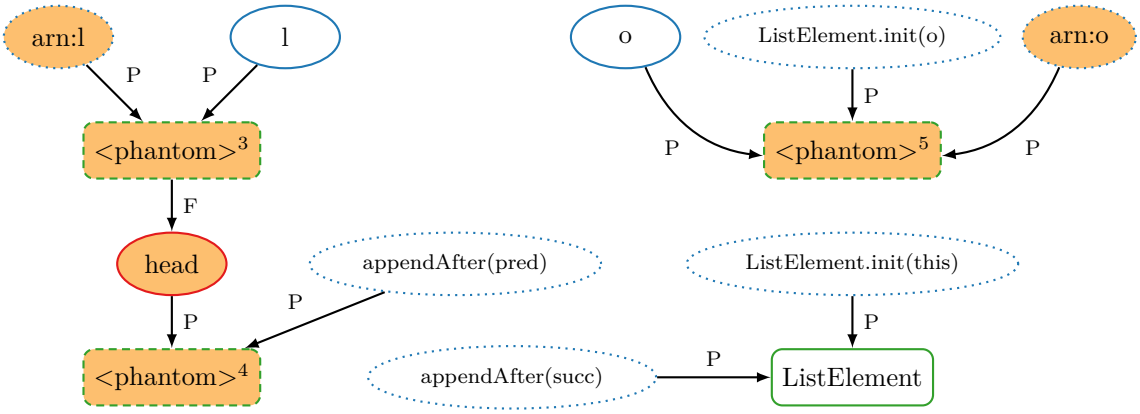


Figure 3.4: The CG of `addElement` as given in Listing 3.3 after intraprocedural analysis. Colors and shapes as described in Figure 3.2.

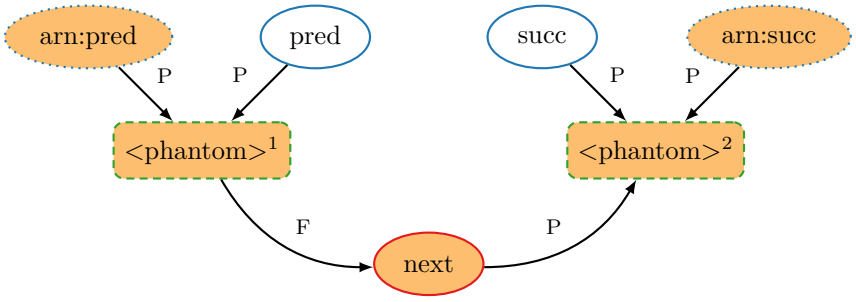


Figure 3.5: The CG of `appendAfter` as given in Listing 3.3 after intraprocedural analysis and path compression. Colors and shapes as described in Figure 3.2.

p	mapsTo(p)
<phantom>^{1}	<phantom>^{4}
<phantom>^{2}	<phantom>^{6}, ListElement

Table 3.2: The *mapsTo* relation for the example given in Listing 3.3

and continue to drill down recursively finding the two phantom object nodes

$$\big(\text{<phantom>}^4, \text{<phantom>}^1\big). \tag{3.4}$$

Since the field edge to the `next` field only exists in the callee CG, the same field is created in the caller's CG. The node reachable from the `next` field reference node corresponds to the `ListElement` object node; however, this information is not available at this point in the analysis, because the invocation's second parameter has not been analyzed yet. For this reason, a phantom object node (henceforth denoted by <phantom>^{6}) is created to represent the pointee of `next`. Since there are no further outgoing edges in the CG of `appendAfter`, analysis of this parameter is finished. Processing restarts at the next pair of parameters, i.e.,

$$\big(\text{appendAfter(succ), arg:succ}\big) \tag{3.5}$$

and will find the tuple

$$\big(\text{ListElement, <phantom>}^2\big). \tag{3.6}$$

Since there are no edges outgoing from the phantom object node, the analysis is complete. While these steps are executed, the algorithm keeps track of a mapping from object nodes in the callee's CG to nodes in the caller's CG. [CGS+03, 4.4.1] calls this the *mapsTo(p)* relation; see Table 3.2 for the *mapsTo* relation in our example. Figure 3.6 shows the intermediate state of the CG for `addElement` after these steps.

Algorithm 1: The *updateCaller* procedure [CGS+03, Fig. 7]

Input : an invocation site
Result: calls *updateNodes* for all pairs of actual reference nodes

```
1 updateCaller (i : invocation)
2 begin
3     foreach (a_callee, a_caller) ∈ ParameterPairs(i) do
4         updateNodes(a_callee, a_caller) ;
```

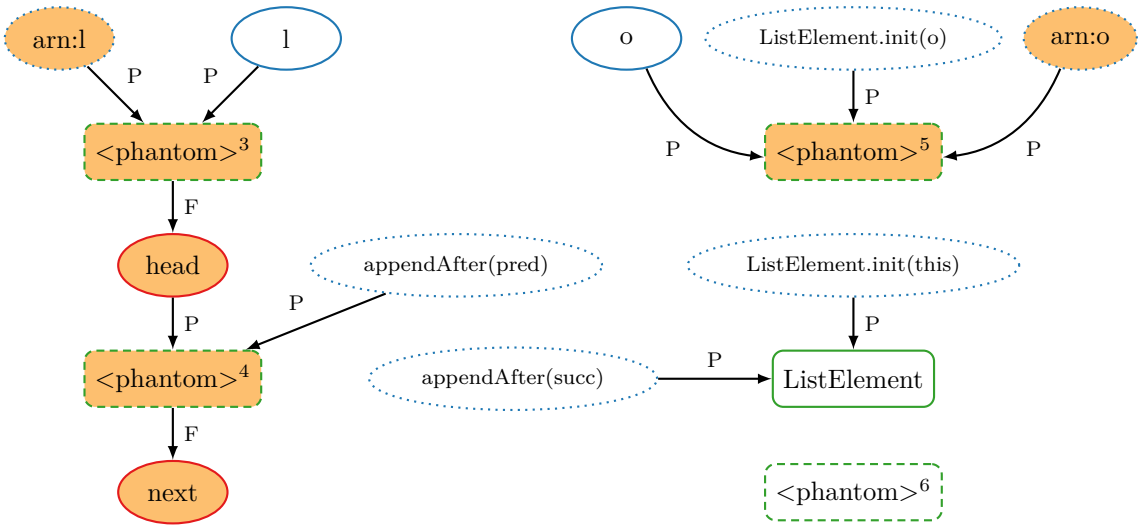


Figure 3.6: The CG of `addElement` as given in Listing 3.3 after *updateNodes* in interprocedural analysis. Colors and shapes as described in Figure 3.2.

These steps are formalized in Algorithms 1 and 2. They ensure all object nodes in the non-local subgraph[10] of the callee's CG are represented in the caller's CG and construct the *mapsTo(p)* relation that will be needed in Section 3.2.2.

Algorithm 1 finds all actual reference nodes in both the caller's and the callee's CGs (see line 3 in Algorithm 1) and calls *updateNodes* for each corresponding pair in statement 4.

The *updateNodes* procedure shown in Algorithm 2 is invoked by *updateCaller* and calls itself recursively. Its parameters are a pair of corresponding reference nodes, denoted by f for the callee side and g for the caller side. The algorithm loops for each object node pointed to by f (statement 3) and ensures these nodes are represented in the caller CG, creating a new node if they are not, in statements 4 to 6. Statements 7 to 10 then ensure all object nodes below g in the caller's CG are present in the *mapsTo(p)* set of the callee object node f. If a node was added to the set, the following instructions are also executed: First, if the escape state of the callee object node is *global*, the escape state of the nodes mapping to this object node in the caller's CG are also marked *global* in line 13. Second, the algorithm makes sure all fields present in the callee's CG below the currently analyzed object node (statement 14) are also present in the caller's CG, creating them if they are not in line 16. Finally, any two corresponding fields found at this level are used to recursively call *updateNodes* in statement 17.

[10]Since processing always starts at callee-side actual reference nodes which are always part of the non-local subgraph, all nodes reachable from these must also be part of the non-local subgraph.

Algorithm 2: The *updateNodes* procedure [CGS+03, Fig. 7]

Input : a pair of corresponding reference nodes on the caller and callee side
Result: ensures all nodes in the callee CG are represented in the caller's CG

```
1  updateNodes(f : reference node, g :
   reference node mapping to f in the caller's CG)
2  begin
3      foreach n ∈ PointsTo(f) do
4          if PointsTo(g) = ∅ then
5              // Make sure the pointees of f are represented in the caller's CG
6              CreateTargetNode(g) ;
7          foreach m ∈ PointsTo(g) do
8              if m ∉ MapsTo(n) then
9                  // Update the MapsTo(n) relation
10                 MapsTo(n) ∪= {m} ;
11                 if Global == EscapeState(n) then
12                     // Nodes globally escaping the callee also escape the caller
13                     EscapeState(m) = Global ;
14                 foreach k ∈ Fields(n) do
15                     // Make sure the fields of n are represented in the caller's
                          CG
16                     callerField = GetField(m, FieldName(k)) ;
17                     updateNodes(k, callerField) ;
```

3.2.2 | Edge Propagation

While the steps in Section 3.2.1 ensure that all nodes in the non-local subgraph of the callee's CG are represented in the CG of the caller, this alone is not sufficient to generate correct information, because edges present in the callee's CG have not been propagated to the caller yet. To do this, we iterate over the object nodes in the callee's non-local CG and call *updateEdges* (see Algorithm 3).

Algorithm 3: The *updateEdges* procedure [CGS+03, 4.4.2]

Input : An object node in the callee's non-local subgraph
Result: Points-to edges present in the callee's CG are added to the caller's CG

```
1 updateEdges(n : callee-side object node)
2 begin
3     foreach k ∈ Fields(n) do
4         foreach m ∈ MapsTo(n) do
5             callerField = GetField(m, FieldName(k));
6             foreach o ∈ PointsTo(k) do
7                 PointsTo(callerField) ∪= MapsTo(o);
```

UpdateEdges operates on a given object node in the callee's CG and starts by getting the field reference nodes reachable from this object node (see statement 3 in Algorithm 3). Lines 4 and 5 compute the equivalent of these field reference nodes on the caller side by using the *mapsTo(p)* property as computed in *updateNodes*. The two following statements in lines 6 and 7 finally propagate any edges present in the callee's CG into the caller's CG.

Applied to the previously discussed example, calling

$$\text{updateEdges}\Big(\text{<phantom>}^1\Big) \tag{3.7}$$

results in the operation

$$\text{PointsTo(next)} \cup= \Big\{\text{<phantom>}^6, \text{ListElement}\Big\}, \tag{3.8}$$

i.e., adding an edge from next to both <phantom>^{6} and ListElement. Since both object nodes are now reachable from a node with an escape state of *method*, applying Equations (3.1) and (3.2) yields a new escape state *method* for both object nodes. Figure 3.7 depicts the CG after applying *updateEdges*.

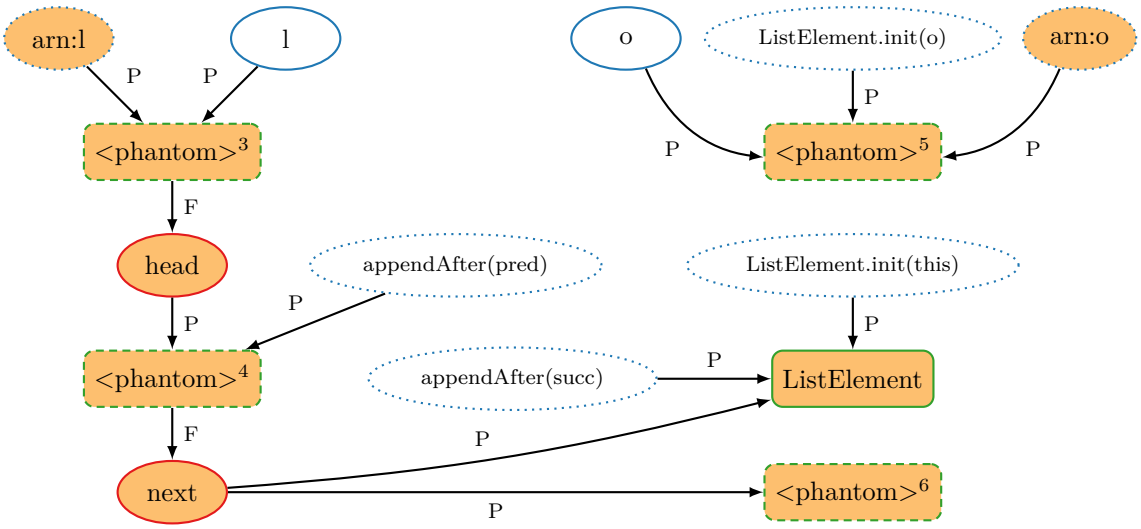


Figure 3.7: The CG of `addElement` as given in Listing 3.3 after interprocedural analysis. Colors and shapes as described in Figure 3.2.

3.3 | Static Stack Allocation

Every allocation site with an escape state of *local* after the analysis in Sections 3.1 and 3.2 is eligible for dynamic stack allocation. Choi et al. suggest using `alloca(3)`, but this function is not standardized and it might not be known how much memory will be allocated on the stack at runtime when using it, e.g., within a loop or when called with a dynamic size. Because `alloca(3)` moves the stack pointer by the number of bytes given as argument, generating code that calls it with a dynamic size or a fixed size but a dynamic number of times (such as in a loop) might overflow the stack. These overflows could possibly endanger the safety of the KESO virtual machines.

For these reasons, statically allocating as many objects as possible on the stack is desirable. However, not all allocation sites with a *local* escape state can always be statically allocated on the stack. This section describes when an allocation can safely be transformed into a static stack allocation.

3.3.1 | Determining Overlapping Liveness Regions

Allocations can only be easily transformed into static stack allocations if there are no two objects allocated by the same statement in use at the same time [CGS+03, 6.2]. To determine whether this is the case, the liveness property of variables can be used

```
1  public class Overlap {
2    Overlap next;
3
4    public static void main(String[] args) {
5      Overlap o = new Overlap();
6      Overlap p = null;
7
8      for (;;) {
9        p = o;
10       o = new Overlap();
11       System.out.println(p);
12       System.out.println(o);
13     }
14   }
15 }
```

Listing 3.4: Example Java code with objects not easily statically stack-allocatable

as follows: Using existing liveness information on basic block level, iterate over the basic blocks in reverse instruction order and carry along a set of variables alive at the current intermediate code instruction. Update this set when the current instruction changes the set of live variables. When encountering an allocating instruction which is a candidate for stack allocation, perform reachability analysis in the CG to determine whether the object allocated is reachable from any of the variables live at this point in the program. If this is the case, multiple objects created at this allocation site are used simultaneously. Since these objects must be distinct and cannot share the same storage location, they cannot trivially be stack-allocated in a static manner[11]. Listing 3.4 gives an example where an allocation (instruction 10), although correctly marked *local*, cannot be turned into a static stack allocation, because the objects allocated by the instruction are reachable from a live variable (p, live starting at instruction 9, ending at 11) at the time of allocation.

3.3.2 | Handling Portals and Native Methods

KESO offers so-called "portal" methods allowing domains to exchange data. Depending on the implementation of this mechanism, passing an object to a portal method could require it to be heap-allocated. However, since KESO creates deep copies of objects passed into portals, the memory pool used to allocate the source object is not relevant. Escape analysis can thus simply ignore portal methods.

[11] If it is possible to determine how many objects allocated by an instruction are alive simultaneously, this number of objects can be statically stack-allocated and managed in a bounded buffer with random access. Every read of one of these objects must be matched to the correct index in the bounded buffer. This is, however, beyond the scope of this thesis.

Native methods, on the other hand, could cause its parameters to escape. To handle this correctly, KESO offers annotations on native methods, allowing the programmer to specify whether parameters passed to a native method escape (e.g., via a global variable). If no annotation is present, the analysis conservatively assumes any object passed to a native method escapes and sets its escape state to *global*.

3.3.3 | Stack Allocation

To turn the allocations identified as *local* into stack allocations, *JINO*'s backend supports emitting C code that will cause objects to be created on the stack rather than using KESO's dynamic memory allocation strategy. To achieve this, *JINO* uses standard C mechanisms and does not rely on `alloca(3)`.

Since KESO's heap object allocation API also initializes the object's meta information, a variant of this API was developed that will initialize these fields from a given pointer to an object. For stack-allocated objects, these functions are called instead of their counterparts allocating from heap.

As part of this thesis, *JINO*'s backend was also modified to support stack-allocating single-dimension arrays of both primitive and complex type. Arrays with constant sizes are allocated on the stack up to a configurable threshold of bytes. Experimental support was also added for stack-allocation of arrays with dynamic sizes, but uses `alloca(3)` and is thus potentially unsafe without further checks.

4 | Evaluation

The previous chapter detailed the design and implementation of new analysis and optimization passes to stack-allocate objects not escaping their context of creation. This chapter compares the effectiveness of the new optimization pass on the basis of measurements against both no similar optimization and the analysis previously implemented as described in Section 2.3.

4.1 | Benchmark CD_x

When evaluating optimizations, benchmarks are a widely used instrument. In order to generate meaningful results a benchmark should be similar to real-world tasks. On the other hand, a benchmark should produce simple numerical output that can easily be processed, compared and graphed. Previous work on the KESO system put an emphasis on the similarity to common real-world applications rather than using a series of micro-benchmarks gauging isolated aspects [Erh11, ESLSP11, STWSP12] by using the CD_j benchmark from the CD_x family of benchmarks first published in 2009 in [KHP+09]. CD_x is "an open-source real-time Java benchmark family that models a hard real-time aircraft collision detection application." [KHP+09] This benchmark consists of two main components: (1) an *air traffic simulator* (ATS) generating a stream of radar frames, which are passed via a non-blocking queue to (2) the *collision detector* (CD), which detects potential collisions among aircraft in these radar frames.

Two variants of this benchmark were used when evaluating the optimization:

1. The "on-the-go" variant generates the necessary radar frames in the collision detection thread as needed for the detection, avoiding the queue at the cost of less realism.

   This variant was tested on a TriCore TC1796 microprocessor running a CiAO system.

2. The "simulated" flavor, where the ATS runs in a concurrent task in a separate

	CD$_x$ on-the-go	**CD$_x$ simulated**
CPU	Infineon TriCore TC1796 150 MHz CPU 75 MHz system	Intel Core i5 650 3.2 GHz
Memory	2 MiB Flash, 1 MiB SRAM	4 GiB DDR3-PC1333
OS	CiAO r1689	Linux 3.2.0
Compiler	GCC 4.5.2, Binutils 2.20	GCC 4.5.3, Binutils 2.22
KESO	r2988	

Table 4.1: Hard- and software configuration used to run the benchmarks.

KESO domain. Frames are passed to the collision detector thread via a queue and are dropped on overflow (i.e., when frames are generated faster than they can be processed).

Since no TriCore port of this variant is available, all measurements were executed on a PC using Trampoline as OSEK abstraction layer.

4.2 | Measurements and Results

See Table 4.1 for the hardware and software configuration used to run the benchmarks. The following sections detail and interpret the measurements and their results conducted.

4.2.1 | Number of Stack Allocations

The number of stack allocation instructions in the generated code can serve as criterion as to the quality of our analysis, since more stack allocations mean less work for the garbage collector. Figure 4.1 graphs the number of objects and arrays that both the old and new algorithm found stack-allocatable and turned into stack allocations relative to the number of allocations in the whole program. In the "on-the-go" test, previously 27.5 % of objects were found to be eligible for stack allocation. This number increased by 25.0 % to 34.4 % with the newly implemented analysis. The bigger "simulated" build configuration shows similar, albeit smaller numbers: 14.2 % were found stack-allocatable by the old algorithm. This thesis' algorithm increased this number by 27.8 % to 18.1 %.

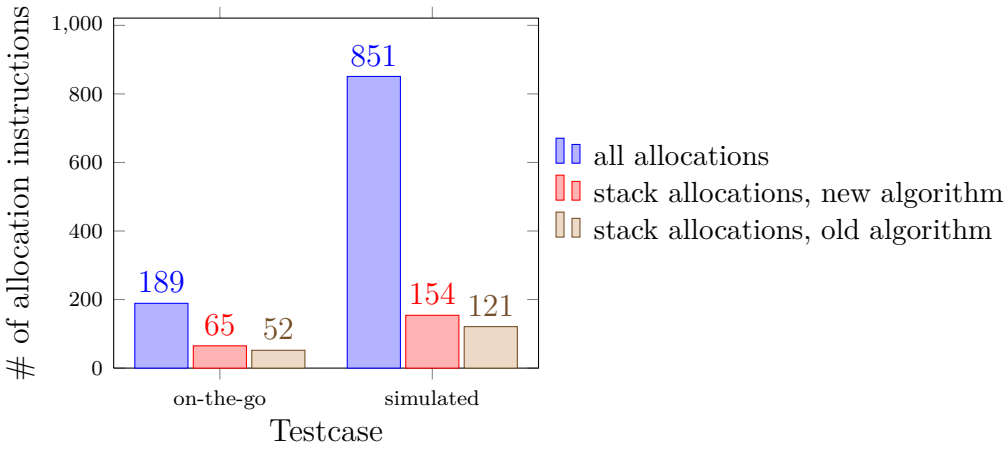


Figure 4.1: Number of stack allocations per test and algorithm relative to all allocating instructions

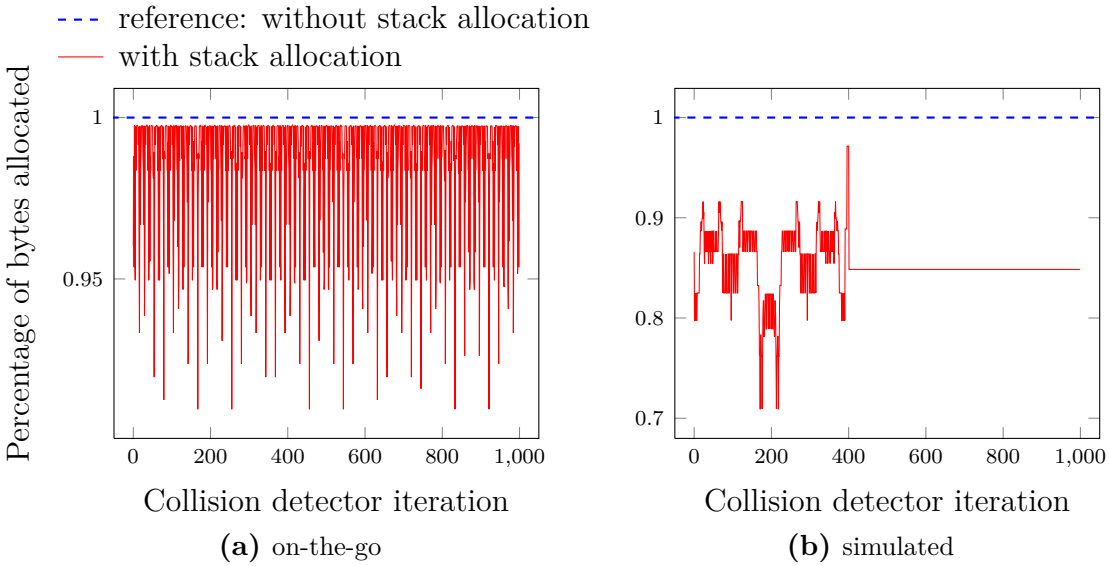
4.2.2 | Amount of Stack-allocated Memory

The more memory is allocated on the stack, the smaller is the strain on the garbage collection methods, because the heap memory does not fill up as quickly. The memory savings can be measured using the CD_x benchmark by comparing the amounts of memory allocated in each run of the collision detector with the same values without using stack allocation. Figure 4.2 depicts the amount of memory used relative to a run without escape analysis for both the "on-the-go" and the "simulated" variant of CD_x. It can be seen that having stack allocation enabled reduces the amount of memory allocated from heap in every case. Savings range from 0.23 % to 9.04 % with a median of 0.30 % in the "on-the-go" variant and 2.86 % up to 29.10 % with the median at 15.15 % in the "simulated" configuration. There is no difference between the old and the new escape analysis in this benchmark, because the objects additionally found to be stack-allocatable are not located in the measured regions of the test.

It is worth noting that the previously implemented analysis benefits from improvements added to the backend in this thesis (most notably the stack-allocation of array types).

4.2.3 | Runtime Savings through Stack Allocation

To answer the question whether stack allocation has an impact on the runtime of the system, the "on-the-go" timing results were compared against those of a run without



(a) on-the-go **(b)** simulated

Figure 4.2: Amount of memory used from heap relative to a run without stack allocation

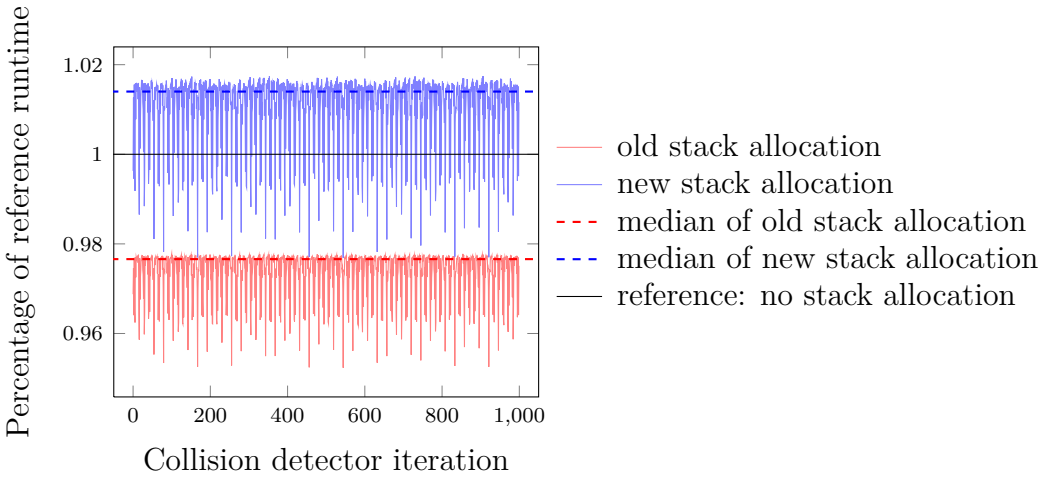


Figure 4.3: Runtime of "on-the-go" relative to a run without stack allocation

stack allocation enabled. See Figure 4.3 for a graphical rendering of these results. Surprisingly, although the measured parts of the code do not differ in both the old and new escape analysis, the old came out ahead with a median speed improvement of 2.34 % compared to the reference run, while the new analysis lost 1.40 % of performance.

The "simulated" test case does not show this slow-down and both the old and new escape analysis algorithm are on par as expected. Figure 4.4 graphs the results, which are scattered compared to the "on-the-go" test, because they were measured on x86 hardware running a Linux operating system, where scheduling effects and other load might have affected the measurement.

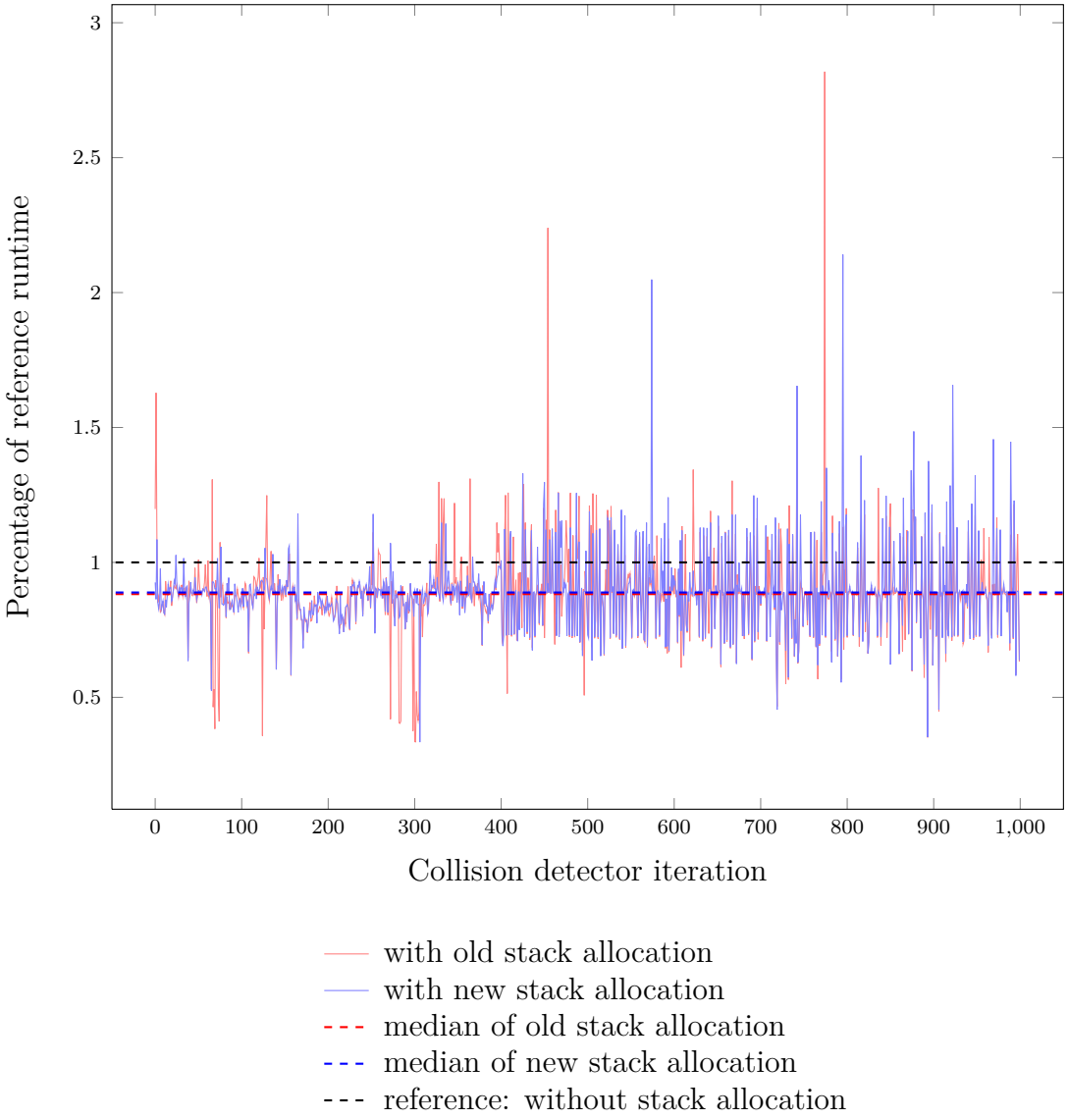


Figure 4.4: Runtime of "simulated" relative to a run without stack allocation

5 | Conclusion and Future Work

In this thesis, design and implementation of an algorithm to automatically convert object allocations from heap to stack allocations where legal was presented and evaluated. This algorithm was written for the middle-end of *JINO*, the KESO Java-to-C compiler in the form of one analysis and one optimization pass. KESO is a Multi-Java virtual machine for statically configured embedded systems.

The analysis was required to be automatic, i.e., it should not require the programmer to decide and specify which objects should be allocated on the stack. Especially applications with intensive dynamic allocation behavior should benefit from the new optimization.

Analysis and optimization are based on a paper implementing stack allocation and synchronization optimizations for Java using escape analysis by Choi et al. first published in 2003 [CGS+03]. The algorithm was re-implemented in *JINO* and adapted to the needs and environment posed by KESO's target domain, statically configured systems for small embedded systems. To simplify the analysis, the existing SSA form was used rather than running the algorithm on non-SSA formed code. This simplified the flow-sensitive part of the analysis. Changes were also required to handle the native code interface available in the KESO API. An analysis suggested in [CGS+03, 6.2] was implemented to determine which objects' liveness periods are non-overlapping and can be statically allocated on the stack, instead of allocating all stack memory using `alloca(3)` at runtime.

Future Work

The results of the analysis can not only be used for the implemented stack allocation. Further possibilities are opened up by the information gathered, such as

transforming more allocations into stack allocations and removing synchronization primitives where superfluous.

Synchronization Optimizations

Apart from stack allocation, [CGS+03] also discussed removing synchronization instructions where unneeded, because the objects used from synchronization are only reachable from a single thread. To achieve this, the escape state of objects is divided into three stages, *global*, *method* and *local*. Objects which are not marked *global* and are used for synchronization are possible targets for this kind of optimization, because there is at most one thread holding a reference to these objects. This means there will be no second concurrent thread attempting to lock the object simultanously, lifting the requirement for expensive atomic synchronization primitives.

It is not clear how much performance could be gained from implementing this. Considering concurrency is not very common in small embedded systems, a large number of objects and synchronized operations in standard library code are possible targets for this optimization.

Stack Allocation in the Caller's Stack Frame

Objects which escape the method of their allocation, but not the thread in which they were allocated, i.e., which have an escape state of *method*, might be eligible for another optimization frequently seen as a pattern in C code: instead of allocating and returning a pointer to memory, these functions require the caller to provide a pointer to a sufficiently large buffer and write their results into this buffer. This leaves the decision on the storage class of the memory to the caller, who could allocate the buffer both on the stack and from heap memory.

Similarly, an object allocated in and returned from a method in Java could be turned into a stack allocation from the caller's stack frame and passed as reference into the called method. Since the `new` operation and the constructor invocation are separate bytecode instructions in Java, the allocation could be moved to the caller, leaving the call to the constructor in place without affecting legality.

Special care needs to be taken as to whether this is legal in situations where a number of methods can call themselves recursively, i.e., when the call graph forms a strongly connected component.

This optimization has the potential to serve another considerable share (starting from 23 % up to 56 % in the benchmarks used in this thesis) of allocations from stack memory, further lifting the strain on garbage collection mechanisms and speed up the execution. Ideally, the use of a garbage collector could be avoided completely.

Bibliography

[CGS+03] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Trans. Program. Lang. Syst.*, 25(6):876–910, November 2003.

[Col12] Bernard Cole. KESO: A Java VM an MCU developer could love? maybe. Jul 2012. `http://www.eetimes.com/electronics-blogs/cole-bin/4389892/KESO--A-Java-VM-an-MCU-developer-could-love--Maybe-`, accessed 2012-08-04.

[Dib06] TimeSys Corp. Dibble, Peter. *The Real-Time Specification for Java 1.0.2*, 2006. `http://www.rtsj.org/specjavadoc/mem_overview-summary.html`, accessed 2012-08-06.

[Erh11] Christoph Erhardt. *A Control-Flow-Sensitive Analysis and Optimization Framework for the KESO Multi-JVM.* Diplomarbeit, Friedrich-Alexander University Erlangen-Nuremberg, March 2011.

[ESLSP11] Christoph Erhardt, Michael Stilkerich, Daniel Lohmann, and Wolfgang Schröder-Preikschat. Exploiting static application knowledge in a Java compiler for embedded systems: A case study. In *JTRES '11: 9th Int. W'shop on Java Technologies for real-time & embedded Systems*, pages 96–105, New York, NY, USA, 2011. ACM.

[GJS+12] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification.* Java SE 7 edition, Feb 2012. `http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf`.

[KHP+09] Tomas Kalibera, Jeff Hagelberg, Filip Pizlo, Ales Plsek, Ben Titzer, and Jan Vitek. CD_x: a family of real-time java benchmarks. In *JTRES '09: 7th Int. W'shop on Java Technologies for real-time & embedded Systems*, pages 41–50, New York, NY, USA, 2009. ACM.

[LLV] The LLVM Project. *Writing an LLVM Pass.* `http://llvm.org/docs/WritingAnLLVMPass.html`, accessed 2012-06-28.

[Ora11] Oracle Corp. *JavaCompiler (Java Platform SE 7)*, Jul 2011. `http://docs.oracle.com/javase/7/docs/api/javax/tools/JavaCompiler.html`, accessed 2012-08-06.

[STWSP12] Michael Stilkerich, Isabella Thomm, Christian Wawersich, and Wolfgang Schröder-Preikschat. Tailor-made JVMs for statically configured embedded systems. *Concurrency and Computation: Practice and Experience*, 24(8):789–812, 2012.

[Tar72] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

List of Figures

List of Tables

List of Algorithms

List of Listings

Acronyms

CG connection graph. 20–26, 28–30, 32–39, 51

JNI Java native interface. 13

LLVM Low-Level Virtual Machine. 18

SSA static single assignment. 18, 19, 23, 25, 28, 47